Notas de Aula
C++

Routo Terada
www.ime.usp.br/~rt
Depto. C. da Computação - USP

Objetivos

- Introdução a C++
- POO - Programação Orientada a Objetos
- Inheritance ou herança
- Constructor e desctructor
- Exemplos de programas

Bibliografia:
1. I. Pohl: C++ for C Programmers, 3rd. edition, Addison-Wesley
2. B. Stroustup: The C++ Programming Language, Addison-Wesley
3. Lee and Phillips: The Apprentice C++, ITP Editors

```
/********************************************************
  Programa:  ALÔ1.cpp
  Object Oriented Programming
  Compilador:  Borland C++     Version 5.01
********************************************************/
//ALÔ em C++

#include <iostream>      //Input/Output library
#include <string>        //tipo string
using namespace std;
inline void pr_message(string s = "ALÔ Mundo!")
{ cout << s << endl; }
int main()
{
   pr_message();
}
```

P/ evitar escrever std:: antes de nomes

Vai exibir Alô Mundo!


```
#include <iostream>        //IO
#include <string>
using namespace std;    // ALÔ2.cpp
inline void pr_message(string s = "Alô Mundo!")
{ cout << s << endl; }

int main()
{
   pr_message();
   pr_message("Pedro Cabral");
   pr_message("É agora?");
}
```

Vai exibir :

Alô Mundo!

Pedro Cabral

É agora?

Palavras-chave em C++

```
and          false        short
and_eq       float        signed
asm          for          sizeof
auto         friend       static
bitand       goto         struct
bitor        if           switch
bool         inline       template
break        int          this
case         long         throw
catch        mutable      true
char         namespace    try
class        new          typedef
compl        not          typeid
const        not_eq       typename
continue     operator     union
default      or           unsigned
delete       or_eq        using
do           private      virtual
double       protected    void
else         public       volatile
enum         register     wchar_t
explicit     return       while
extern                    xor
                          xor_eq
```


- Um tipo agregado permite um conjunto de dados componentes ser tratado como um único tipo de dado
  - enquanto permite o acesso aos componentes individuais
- A construção struct de C++ permite a definição de tipos agregados
- O acesso às variáveis-membro é através do operador ponto ".", como em x.peso

Nome de tipo novo

Variáveis-membro

```
struct Identificador_de_tipo
{
   Tipo var_membro1;
   Tipo var_membro2;
   ...
   Tipo var_membroN;
};
```

PARADIGMA DE ORIENTAÇÃO A OBJETOS - POO

- C++ é uma linguagem baseada no paradigma de orientação a objetos - POO (assim como Java)
- Este paradigma tem várias características marcantes:
  - Alta reutilização de software (via class)
  - Desenvolvimento por aprimoramentos sucessivos
  - Construções complexas a partir de construções simples (via class)
  - Chance maior de se ter manutenção simples


```
#include <iostream>
#include <string>
using namespace std;
struct ponto{
   double x, y; //coordenadas x e y de um ponto
   void plus(ponto c);       //protótipo de função
   void print(string nome);
   void print() { cout << "(" << x << "," << y << ")"; }
   void init(double u, double v) { x = u; y = v; }
};
void ponto::print(string nome)
{
   cout << nome << " (" << x << "," << y << ")";
}
void ponto::plus(ponto c)     //definição não é inline
{
//`offset' do ponto existente pelo ponto c
   x += c.x;
   y += c.y;
}
int main()
{
   ponto w1, w2;
   w1.init(0, 0.5);
   w2.init(-0.5, 1.5);
   cout << "\nponto w1 = ";
   w1.print();
   cout << "\nponto w2 = ";
   w2.print();
   w1.plus(w2);
   cout << "\nponto w1 depois de plus = ";
   w1.print();
}
```

Funções são membros de struct em C++

:: scope resolution operator

Overloading: w1.print(“ponto w1 = “);

Vai exibir:
ponto w1 = (0 , 0.5)
ponto w2 = (-0..5 , 1.5)
ponto w1 depois de plus = (-0.5 , 2.0)

```
#include <iostream>
#include <string>        // ponto2.cpp
using namespace std;
struct ponto {
public:
   void print(){ cout << "(" << x << "," << y << ")";}
   void init(double u, double v) { x = u; y = v; }
   void plus(ponto c);
private:
   double x, y;
};
void ponto::plus(ponto c){      //definição não é inline
//`offset' ponto existente pelo ponto c
   x += c.x;
   y += c.y;
}
int main(){
   ponto w1, w2;
   w1.init(0, 0.5);
   w2.init(-0.5, 1.5);
   cout << "\nponto w1 = ";
   w1.print();
   cout << "\nponto w2 = ";
   w2.print();
   w1.plus(w2);
   cout << "\nponto w1 depois de plus = ";
   w1.print();
}
```

Só pode ser usado dentro deste struct (private)

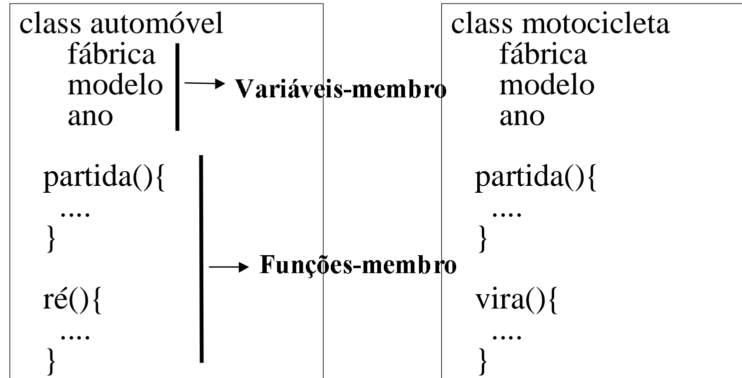
Vai exibir:
ponto w1 = (0 , 0.5)
ponto w2 = (-0..5 , 1.5)
ponto w1 depois de plus = (-0.5 , 2.0)

classes

- **Necessitamos de um mecanismo para definir tipos de dados que**
  - ✔ **separa claramente a interface do tipo da sua implementação (*separation of concerns* e *information hiding*)**
  - ✔ **previne "abuso" dos detalhes da implementação (*encapsulation*)**
  - ✔ **permite definição de comportamento (I.e. funções), assim como valores abstratos**
  - ✔ **permite verificação de erros durante a compilação**
  - ✔ **permite desenvolvimento separado dos componentes de tipo de dados**
- **O mecanismo de `class` em C++ permite satisfazer estes requisitos**

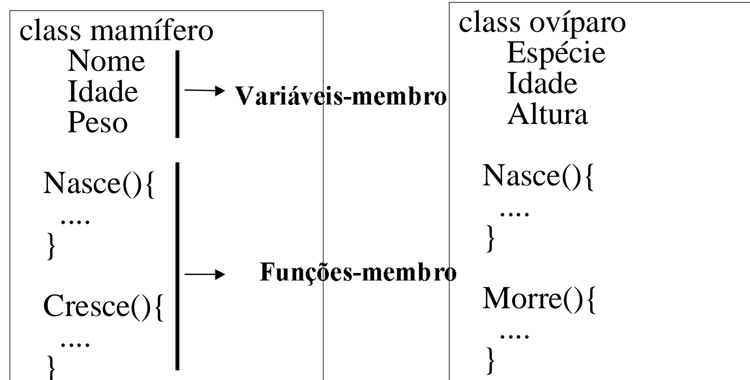
CLASSES

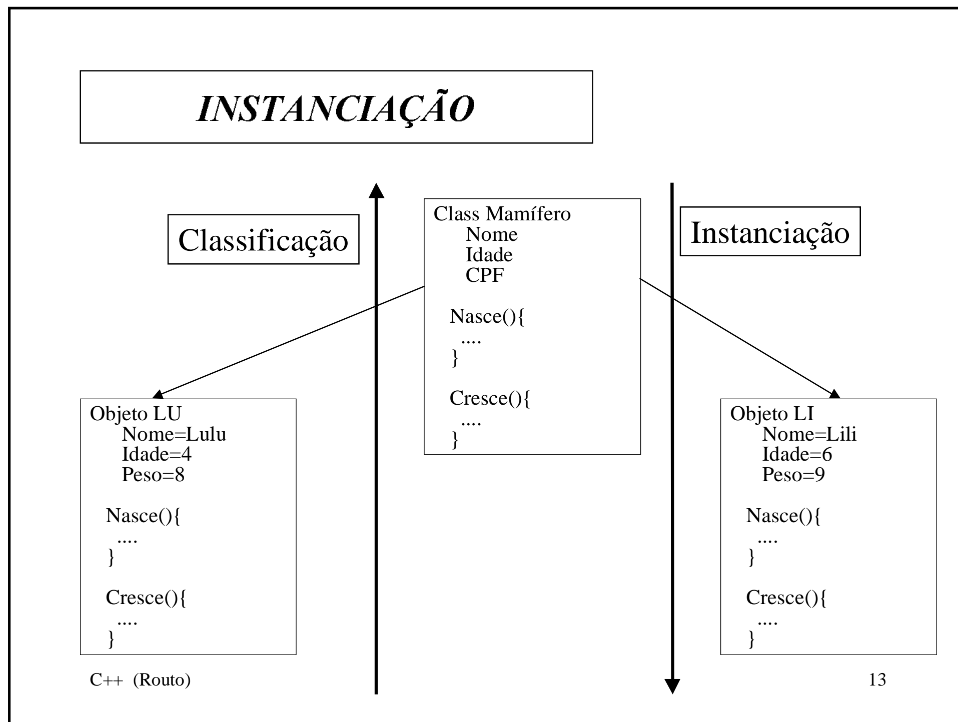
São abstrações de conjuntos de objetos

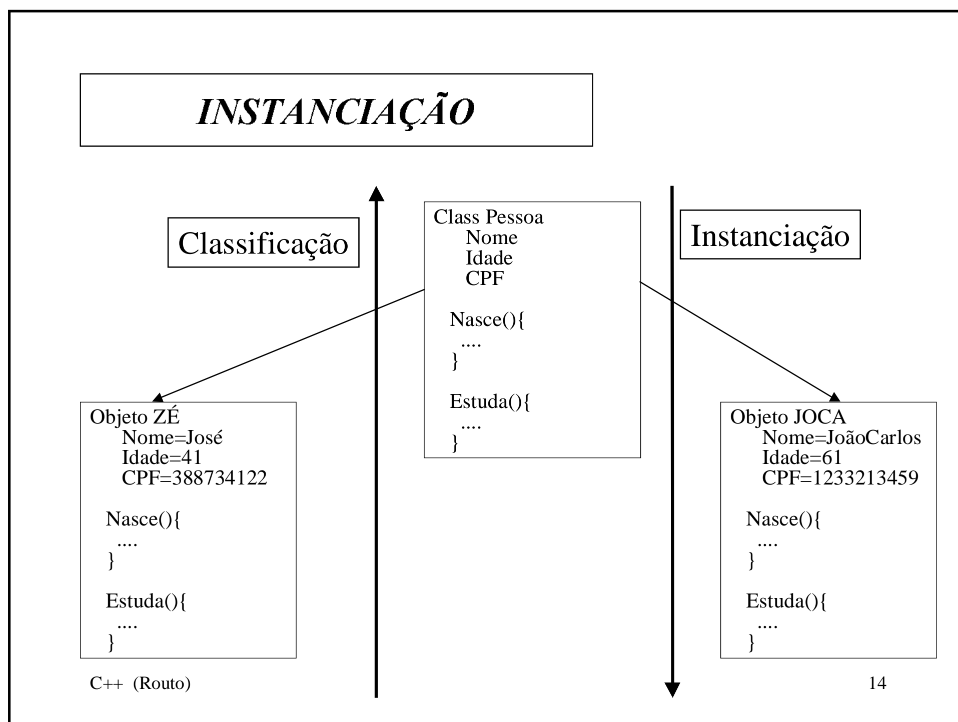


CLASSES

São abstrações de conjuntos de objetos




INSTANCIAÇÃO
Classificação
Instanciação
Class Mamífero
Nome
Idade
CPF
Nasce(){
....
}
Cresce(){
....
}
Objeto LU
Nome=Lulu
Idade=4
Peso=8
Nasce(){
....
}
Cresce(){
....
}
Objeto LI
Nome=Lili
Idade=6
Peso=9
Nasce(){
....
}
Cresce(){
....
}
C++ (Routo)
13


INSTANCIAÇÃO
Classificação
Instanciação
Class Pessoa
Nome
Idade
CPF
Nasce(){
....
}
Estuda(){
....
}
Objeto ZÉ
Nome=José
Idade=41
CPF=388734122
Nasce(){
....
}
Estuda(){
....
}
Objeto JOCA
Nome=JoãoCarlos
Idade=61
CPF=1233213459
Nasce(){
....
}
Estuda(){
....
}
C++ (Routo)
14

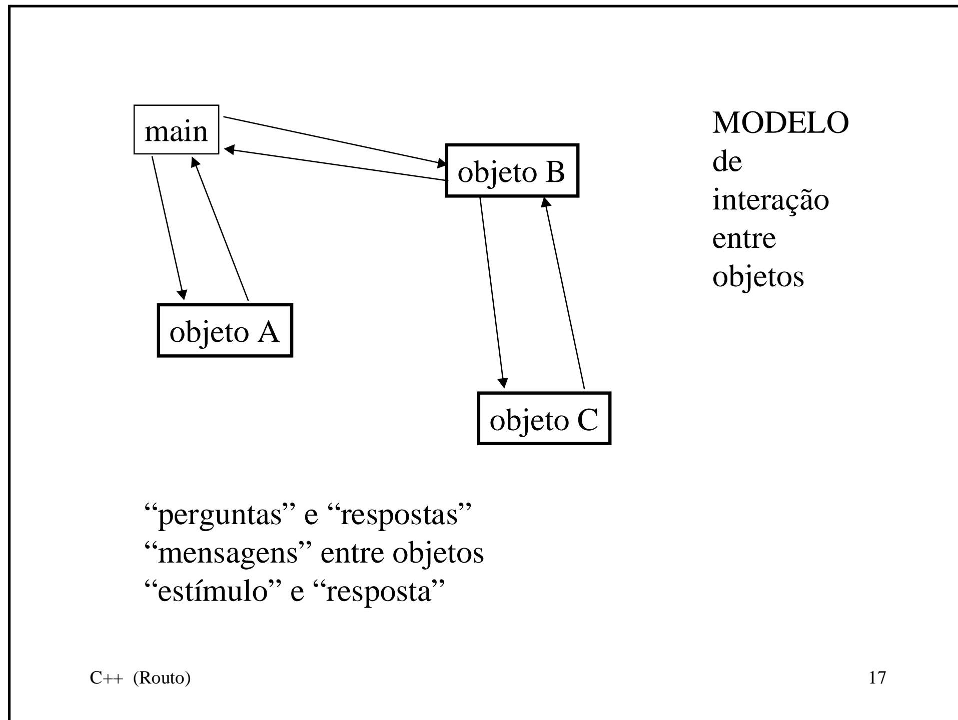
Palavras-chave

- OBJETOS
- CLASSES
- INSTANCIAÇÃO

OBJETOS

Um objeto (como LU ou ZÉ) é caracterizado por:

- Uma identificação (*handle*)
- Um estado interno (variáveis-membro)
- Um comportamento (funções-membro)



class

ponto4.cpp

```
#include <iostream>
#include <string>
using namespace std;
class ponto {
   double x, y;         //implicitamente `private'
public:
   void print() { cout << "(" << x << "," << y << ")"; }
   void init(double u, double v) { x = u; y = v; }
   void plus(ponto c);
};
void ponto::plus(ponto c) //definição não é inline
{
//`offset' ponto existente pelo ponto c
   x += c.x;
   y += c.y;
}
int main()
{
   ponto w1, w2;
   w1.init(0, 0.5);
   w2.init(-0.5, 1.5);
   cout << "\nponto w1 = ";
   w1.print();
   cout << "\nponto w2 = ";
   w2.print();
   w1.plus(w2);
   cout << "\nponto w1 depois de plus = ";
   w1.print();

}
```

Vai exibir:
ponto w1 = (0 , 0.5)
ponto w2 = (-0..5 , 1.5)
ponto w1 depois de plus = (-0.5 , 2.0)

C++ (Routo) 18

```
#include <iostream>
using namespace std;  //ponto5a.cpp

class ponto {
public:                       //indique membros `public' primeiro
void   print() { cout << "(" << x << "," << y << ")"; }
   void   init(double u, double v) { x = u; y = v; }
   void   plus(ponto c);
   ponto  inverso() { x = -x; y = -y; }  //retorna ponto
private:                      //explicitamente private
   double x, y;
};

int main()
{
   ponto  a, b;
   a.init(1.5, -2.5);
   a.print();                 // (1,5 , -2.5)
   b = a.inverso();
   b.print();                 // (-1.5 , 2.5)
}
```


- **Uma definição de class define os detalhes exatos em C++ do que a interface com a classe é, e desempenha dois papéis**
  - **papel de facilitar verificação de erros em tempo de compilação**
  - **papel de documentação - detalhes exatos de como usar uma classe**
- **Definições de class possuem a estrutura geral seguinte:**

```
class IdTipo
{
  public:   // Parte pública de uma classe;
            //   definição disponível a um usuário
            // Interface "constructor"
    IdTipo();
            // Interfaces de função-membro
    TipoResultado FunçãoMembro(...);
            // Interfaces de operador-membro
    TipoResultado operator+(Tipo lado-direito);

  private:  // Parte privada de uma classe
    ...     // Declarações, def. de tipo e
            //   interfaces de função:
            //   NÃO disponível para um usuário
};
```

Linguagens Orientadas a Objetos

Existem várias linguagens orientadas a objeto, algumas com características bem distintas. Algumas mais familiares são

- C++
- Java
- Object Pascal (Delphi)

Outras menos familiares:
- Eiffel
- SmallTalk


```
#include <iostream>      //Input Output
#include <string>        //tipo string
using namespace std;                     //complexo1.cpp
class complexo {
public:                    //acesso universal
   void re_assign(double r) { real = r; }
   void im_assign(double im) { imagin = im; }
   void print() const
      { cout << "(" << real << ","
             << imagin << "i)" << endl; }
   friend complexo operator+(complexo, complexo);
private:                   //acesso restrito à
                           //implemetação
   double real, imagin;
};

//overloading do operador +
complexo operator+(complexo x, complexo y)
{
   complexo t;
   t.real = x.real + y.real;
   t.imagin = x.imagin + y.imagin;
   return t;
}
```

A palavra "friend" indica que apesar de ser "public", o operador tem acesso à parte "private"

```
int main()
{
    complexo x, y, z;
    x.re_assign(9.5);
    x.im_assign(-4.5);
    y.re_assign(4.2);
    y.im_assign(6.0);
    z = x + y;
    x.print();
    y.print();
    z.print();
}
```

Exibe:
(9.5,-4.5i)
(4.2,6i)
(13.7,1.5i)

Constructor e destructor

```
#include <iostream>      //IO
#include <string>
using namespace std;     //complexo2.cpp
class complexo {
public:                   //acesso universal
   //constructor
   complexo(double r=0, double im=0): real(r), imagin(im) { }
   //destructor
   ~complexo() { cout << "destructor invocado sobre "; print(); }
   void re_assign(double r) { real = r; }
   void im_assign(double im) { imagin = im; }
   void print() const
      { cout << "(" << real << ","
             << imagin << "i)" << endl; }
   void  print(string var_name) const
      { cout << var_name << " = "; print(); }
   friend complexo operator+(complexo, complexo);
private:                  //acesso restrito
double real, imagin;
};
//overloading do operador  +
complexo operator+(complexo x, complexo y)
{
   complexo t;
   t.real = x.real + y.real;
   t.imagin = x.imagin + y.imagin;
   return t;
}
```

```
int main()
{
   complexo  x(5.5, 1.0), y, z;
   complexo w(1.5,2);
   w.print("w");              //print: w = (1.5,2i)
   w.print();                 //print: (1.5,2i)
   x.print("x");
   y.re_assign(4.2);
   y.im_assign(6.0);
   z = x + y;                  //usa operator+()
   x.print("x");
   y.print("y");
   z.print("z");
}
```

Constructor é invocado

y=(0,0), z=(0,0)

Destructor é invocado qdo var. deixa de existir (ou com “delete &x;”)

Overloading de print

```
#include <iostream>      //Input Output
#include <string>        //tipo string
using namespace std;                    //complexo1.cpp
class complexo {
public:                    //acesso universal
   void re_assign(double r) { real = r; }
   void im_assign(double im) { imagin = im; }
   void print() const
      { cout << "(" << real << ","
             << imagin << "i)" << endl; }
   void print(string nome_var) const
      { cout << nome_var << " = "; print(); }
private:                   //acesso restrito à
                           //implemetação
   double real, imagin;
};

int main()
{
   complexo x(1.5,2);
   x.print("x");          //print: x = (1.5,2i)
   x.print();             //print: (1.5,2i)
}
```

Overloading de +

```cpp
#include <iostream>
#include <string>
using namespace std;    //complexo3.cpp
class complexo {
public:                    //acesso universal
   //constructor
   complexo(double r=0, double im=0): real(r), imagin(im) { }
   //destructor
   ~complexo() { cout << "destructor called on "; print(); }
   void re_assign(double r) { real = r; }
   void im_assign(double im) { imagin = im; }
   void print() const
      { cout << "(" << real << ","
            << imagin << "i)" << endl; }
   void  print(string var_name) const
      { cout << var_name << " = "; print(); }
   friend complexo operator+(complexo, complexo);
private:                   //acesso restrito
   double real, imagin;
};
//overloading do operador +
complexo operator+(complexo x, complexo y)
{
   complexo t;
   t.real = x.real + y.real;
   t.imagin = x.imagin + y.imagin;
   return t;
}
```

```cpp
int main()
{
   complexo x(9.5, -4.5), y(4.2,6.0), z;
   z = x + y;
   x.print("x");
   y.print("y");
   z.print("z");
}
```

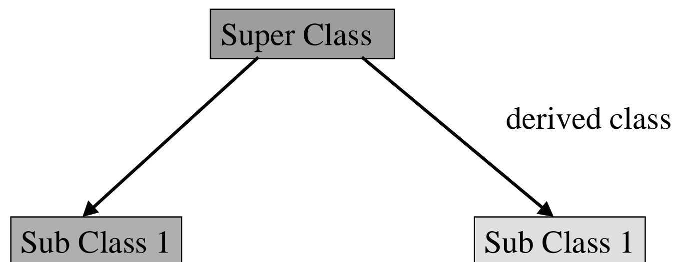
Exibe:

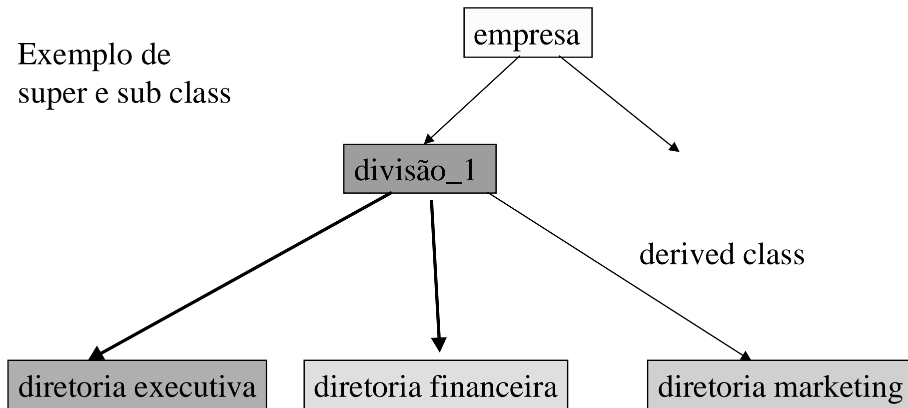
x = (9.5,-4.5i)

y = (4.2,6.0i)

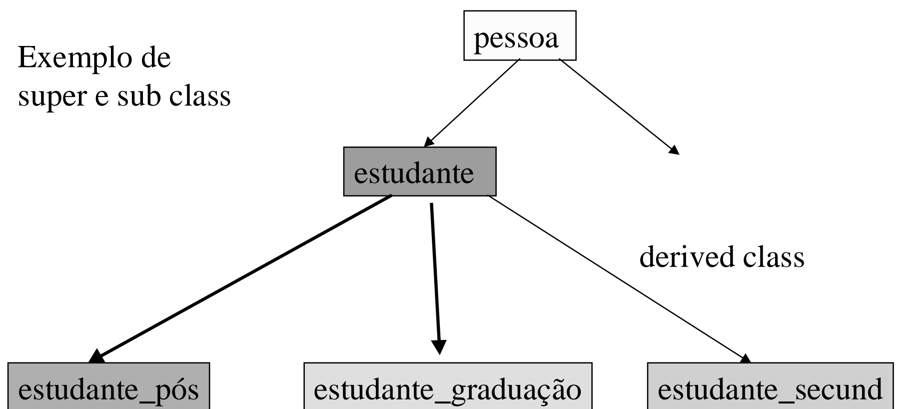
z = (13.7,1.5i)

C++
super e sub class





vantagens:
1. objetos do tipo *diretoria* usam funções já definidas para *divisão_1*;
2. herarquia reflete a relação dos objetos no domínio do problema



vantagens:
1. objetos do tipo *estudante_pós* usam funções já definidas para *estudante*;
2. herarquia reflete a relação dos objetos no domínio do problema

Inheritance ou herança
com *private members*

```
#include <iostream>  //estudante1.cpp
enum ano{ primeiro,segundo,terceiro,quarto,pósgra}
enum bolsa { CNPq, FAPESP, CAPES, outra };

class estudante { //super-class
public:        //constructor de estudante
  estudante(char* enm,int eident,double ent,ano ex);
  void  print() const;
private:
   int     eident_estudanteP;
   double  enotaP;
   ano     eaP;
   char    enomeP[30];
};

class estudante_pós :public estudante{ //sub-class
public:       //constructor de estudante_pós
  estudante_pós(char* enm,int eident,double ent,
       ano ex,bolsa epbls,char* epdpt,char* eptes);
  void print() const;
private:
   bolsa    epbP;
   char     epdeptoP[10];
   char     epteseP[80];
};
```

subclasse: herança definida por :

variáveis adicionais só da subclasse

estudante é *super-classe*

***Inheritance* ou herança**
com *protected members*

```
class estudante {  //estudante2.h
public:
 enum ano{primeiro,segundo,terceiro,quarto,pósgra };
 estudante(char* enm, int eident, double ent, ano ex);
 void  print() const;
protected:
 int     eident_estudanteP;
 double  enotaP;
 ano     eaP;
 char    enomeP[30];
};                     //constructor de estudante
estudante::estudante(char* enm, int eident, double ent,
 ano ex):eident_estudanteP(eident), enotaP(ent),
 eaP(ex)
{ strcpy(enomeP, enm);}
void estudante::print() const
{ cout << enomeP << " , " << eident_estudanteP
        << " , " << eaP << " , " << enotaP << endl;}
```

protected: acessível às sub-classes; exceto isso, é como ser private

estudante_pós é *subclasse* de estudante

```
class estudante_pós : public estudante {
public:
   enum bolsa{FAPESP, CNPq, CAPES, outra};
   estudante_pós(char* enm, int eident, double ent, ano ex,
         bolsa epbls, char* epdpt, char* eptes);
   void  print() const;
protected:
   bolsa     epbP;
   char      epdeptoP[10];
   char      epteseP[80];
};

estudante_pós::estudante_pós(char* enm,int eident,double ent,ano ex,bolsa epbls,
   char* epdpt, char* eptes):estudante(enm, eident, ent, ex), epbP(epbls)
{
   strcpy(epdeptoP, epdpt);
   strcpy(epteseP, eptes);
}

void estudante_pós::print() const
{
   estudante::print(); //exibe info da super class
   cout << epdeptoP << " , " << epbP << '\n'
        << epteseP << endl;
}
```


main() com inheritance

```
#include <iostream>
#include <string>
//Teste das classes estudante e estudante_pós
#include  "student2.h"     //include das declarações de super e sub class
int main(){
estudante  eSilva("Maria Silva", 100, 3.4, estudante::primeiro),
     *peSilva = &eSilva; //pointer
estudante_pós  epPereira("Mário Pereira", 200, 3.2,
              estudante::pósgra, estudante_pós::FAPESP, "Computação",
                     // ano ^^              bolsa ^^    depto ^^
              "Inteligência no Computador"),
                     //  tese ^^
     *pepPereira;  //pointer
   peSilva -> print();          //estudante::print
   peSilva = pepPereira = &epPereira;
   pepPereira -> print();          //estudante_pós::print
   peSilva ->print();           //estudante::print
}
```

exibe:

```
Maria Silva , 100 , 0 , 3.4
Mário Pereira , 200 , 4 , 3.2
Computação , 0
Inteligência no Computador
Mário Pereira , 200 , 4 , 3.2
```

bolsa FAPESP

```
#include <iostream>
#include <string>
using namespace std;
// cálculo de máximo divisor comum -- mdc
int mdc(int m, int n)       //def. funcão
{                           //bloco
   int  r;                  //declara resto
   while (n != 0) {         //não igual
      r = m % n;            //operador modulo
      m = n;                //atribuição
      n = r;
   }                        //fim de laço while
   return m;                //sai de mdc com valor m
}

int main()
{
   int  x, y, g;
   cout << "\nPROGRAMA MDC em C++";
   do {
      cout << "\nDigite dois inteiros: ";
      cin >> x >> y;
      cout << "\nMDC(" << x << ", " << y << ") = "
           << (g = mdc(x, y)) << endl;
      } while (x != y);
}
```

exibe:
MDC(48,56)=8


```
#include <iostream>
#include <assert>
//Uso de new e alocação dinâmica de array
int main()
{
   int*  dado;
   int   tam;
   cout << "\nDigite tamanho do array: ";
   cin >> tam;
   assert(tam > 0);
   dado = new int[tam];          //aloca um array de ints
   assert(dado != 0);            //dado != 0 sucesso na alocação
   for (int j = 0; j < tam; ++j)
      cout << (dado[j] = j) << '\t';
   cout << "\n\n";
   delete[] dado;                //dealoca um array
}
```

exibe:
0 1 2 3 4